MASTER IN COMPUTER SCIENCE

Regression Testing in Software Product Lines

Beeknoo, Kirti

Award date:
2021

Awarding institution:
University of Namur

Université de Namur
Faculté d'informatique
Année académique 2021–2022

Regression Testing in Software Product Lines

Kirti Maushmi Beeknoo

Promoteurs : _____ (Signature pour approbation du dépôt - REE art. 40)
Dr. Gilles Perrouin & Prof. Pierre-Yves Schobbens

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Abstract

Software product lines is a strategic choice for structuring many large-scale software today to support systematic reuse while reducing development costs.

One key challenge for software product lines is to efficiently manage variability throughout their life-cycle while avoiding any regression on untouched code. In this thesis, we address the problem of regression testing in software product lines during their evolution (e.g., addition of functionality, modification or deletion of code). Hence, we propose an approach to the problem and illustrate it using the classical example of the vending machine. Indeed, we implemented the example in a feature model on FeatureIDE and made it to evolve. Then, using a regression testing tool, EvoSuiteR, running generated tests on both the evolved software product line and the original software product line, we were able to successfully generate valid regression tests for the original software product line. (In each case, we have tested the evolved feature models for regression and were able to find a sample test suite for the purpose).

This paper therefore demonstrates a valid practical approach for regression test generation in software product lines. Finally, we give preliminary results in the assessment of our regression test generation method.

Acknowledgements

I would like to thank all the people who contributed to the success of my master thesis despite the sanitary conditions which are not optimal.

First of all, I would like to show my deep gratitude to my master's thesis supervisor Dr. Gilles Perrouin for his patience, his availability and his judicious advice, which contributed to my reflection.

I wish to pay special regards to Prof. Pierre-Yves Schobbens for his tips and for answering my questions. They have been of great support in the development of this master thesis.

I am sure that this research work will contribute to a better understanding of the field for further progress.

List of Tables

List of Figures

Contents

Chapter 1

Introduction

In this chapter, we explain the context of the research work. We give a brief overview of the research problem and the approach proposed.

1.1 Context

Software product lines is a growing concept that many industries have adopted to ensure a low cost of production while maintaining multiple variants of their software to address distinct consumers needs[Ape+13].

Therefore, software product lines is a complex software with many features composing it. A valid combination of those features is a product configuration. In a real-life example of a software product line, there can be a large number of possible product configurations. Owing to this large number, we might need a lot of resources and computing power to address the issue of testing and regression testing in case of evolution on the software product line. This makes the testing phase tedious and costly.

Hence, in our research work, we have proposed an approach to regression testing so as to reduce the cost of regression testing in software product lines.

1.2 Research Problem

The objective of this research work is to generate regression test suites for software product lines. Though this work can be manually done, however, as previously mentioned, it can be tedious and error-prone to create the regression test suite manually for each impacted product configuration.

Hence, the objective of this research work would be to find a way of generating those regression test cases with less effort.

1.3 Overview of the approach

In order to achieve the objective set in 1.2, we look at the available offers from the literature to find tools that will model the software product line and generate the regression test cases automatically. After analysis, we choose the best tool and use it for the experiment in the case of a software product line.

1.4 Thesis structure

This thesis is divided into 5 chapters. Chapter 1 gives an overview of the context of this research. It introduces some main concepts. Chapter 2 gives an extensive overview of software product lines and some known testing strategies. Chapter 3 details the approach proposed to regression testing on software product lines along with the advantages and disadvantages. Building on this, chapter 4 describes the experiments designed to concretely prove how regression testing can be done for software product lines. Chapter 5 focuses on the conclusion of the experiments and the future work that can continued on this topic. The whole work is accompanied by a glossary defining the terms used throughout the work.

Chapter 2

Background information

In this chapter, we introduce the concepts of software testing, software product lines and widely known testing strategies for software product lines. We also explain the core concept of this thesis which is regression testing in software product lines.

2.1 Software testing

Software testing has been an integral part of the software development lifecycle ever since the dawn of software engineering. It has been crucial to the sustainability of software in the long term. Indeed, its main goal is to reduce to the maximum the number of faults in the software, leading to the maximal reduction of failures in production. Some other advantages of software testing are:

- Reduced costs of maintenance,
- Quality and good performance of the software,
- Security from malicious attacks.

2.1.1 Types of software testing

Software testing has multiple aspects. It has been established that there are mainly 3 types of testing [HC15] as listed below:

- Functional testing: Unit testing the code to ensure maximum code coverage and following that with integration testing to ensure that the functionality is working well as a whole. Regression testing is also a form of functional testing whereby the goal is to ensure that the previous untouched code is not affected despite some evolution in the new version.
- Non-functional testing: E.g. Performance testing or any other non-functional requirements testing. That is, for instance, measuring the software response times compared to its specifications.
- Security testing: One form of security testing is penetration testing where the tester tries to avoid the security features and gain access to the assets.

Frequently, generating these tests is time consuming, especially in the case of software product lines where the number of product configurations is potentially very large. In this case, there is a strong need for a testing strategy as listed in 2.3 to avoid a budget and resource problem when large software product lines are involved. Moreover, tools generating regression tests such as Evosuite have been proposed but have never addressed the problem of regression testing for software product lines.

2.1.2 Evosuite

In this section, we introduce the tool EvoSuite used in the experiments of this thesis.
EvoSuite is a tool that automatically generates unit test cases written in Java code only [Fra21]. Faults can be automatically detected if they lead to program crashes, deadlocks, or violate a formal specification. However, in practice, it is rarely the case. Also, test generation comes with a known oracle problem. That is, knowing at generation time if the outputs of the test cases are the expected ones. Evosuite helps the programmer/tester to produce a high code coverage test suite, while tackling the oracle problem by generating a minimum of test assertions the programmer will need to check by himself later on. Indeed, the tests are minimized such that only the ones contributing to achieving the highest coverage are retained. In order to achieve this later advantage, EvoSuite adopts a search-based approach integrating state-of-the-art techniques such as hybrid search[HM10], dynamic symbolic execution[GKS05] and testability transformation[Har+04] [FA14].

It is a tool that can be configured to our needs based on several parameters such as choice of lines, branches, outputs and/or mutation coverage criteria.

EvoSuite is freely available, and can be used on the command line, as a plugin to the Eclipse, IntelliJ development platform or through a web interface. In EvoSuite, whole test generation is used and thus there is no issue of having a collateral coverage criterion. Some of the default test coverage goals identified in EvoSuite are as listed below:

- Line: checks how many statements have been covered where a statement is a line of code.
- Branch: checks if all branches of each conditional statement were covered.
- Exception: checks if the exclusions (described by exceptions in the program) are well covered.
- Weak mutation: checks if slight changes in the code, commonly known as mutations, can be detected by a test case at the moment the test case runs the mutated statement.

In the tool, if there is a need to isolate one of the test coverage goals as listed above, then a simple change such as

```
$EVOSUITE -class tutorial.Person -criterion branch
```

in the command line is required. Since, by default, EvoSuite takes into account all the above listed goals and creates an overall coverage out of them, in order for

the user to understand the efficiency of the parameters chosen in the command-line. Also when the tool is run on a project, the statistics of the test coverage goals can be seen in the folder `evosuite-report/statistics.csv`. Both the test coverage goals and statistics can be modified according to the requirements. This feature makes EvoSuite a flexible tool.

In the context of the thesis, we will describe in 4 how FeatureIDE and EvoSuite have been used.

2.2 Software Product Lines

Some 30 years ago, software used to be handcrafted and tailor-made to the needs of the end-user. This used to be a labour-intensive and costly task despite being customised solely for the end-user.

Over the years and with advancements in the software industry, software experts have realised that they could bring together most of the requirements in a domain since they are mostly the same and standardise the software produced. Indeed, this reduced the costs of production and increased the profit margins. However, they did not fully meet the needs of the end-user. Hence, came software product lines into play [MP14]. It was an in-between solution so that companies could benefit from the cost-effectiveness of standardisation and at the same time provide some flexibility to the end-product by allowing some customisation of the software. The overall goal of software product lines is to capitalize on the development effort put into common features, in several products. Trade-offs are then to be made between the cost of development for making the solutions more generic, and the gains of less development effort to derive new variant.

This new approach to software engineering meant that the whole testing approach had to be adapted. We shall talk more about the testing approaches in software product lines in the section 2.3 later in this chapter. Some companies that have benefited from this type of software approach are [Con21]:

- Bosch,
- Hewlett Packar,
- General Motors.

With the advent of software product lines, methods of requirements analysis to implementation also had to be adapted.

Indeed, software product lines require a specific way of handling their complexities such as variability and the systematic reuse of implementation artefacts. Hence, the need for a well-designed framework. In the software product lines life-cycle, there are two distinct areas:

- Domain engineering: is the study of the expert area that will be implemented in software product lines. It results in finding the commonalities amongst all the products to maximise reuse. In this phase, the software product lines is mapped on a diagram to allow readers to distinguish the commonalities and variabilities [MBC09].

- Application engineering: comprises of developing products for the end-user with the sole purpose of meeting his/her requirements. Taking into consideration the commonalities identified from domain engineering, the products are implemented as in traditional software engineering ways.

Each area, in turn, is divided in 2 spaces namely, the problem space and the solution space:

- Domain implementation: As the name implies, this space is mostly about developing the features identified.
- Product derivation: This space is about combining the different features to verify and validate the product against the requirements of the end-user.

There exists a bilateral communication between the 4 quadrants as explained below: 2.1

FROM	TO	INFORMATION FLOW
Domain analysis	Domain realisation	Feature diagram
Requirements analysis	Product derivation	Feature selection
Domain analysis	Requirements analysis	New requirements and Features
Domain realisation	Product Derivation	Commonalities

Table 2.1: Communication in the SPL framework

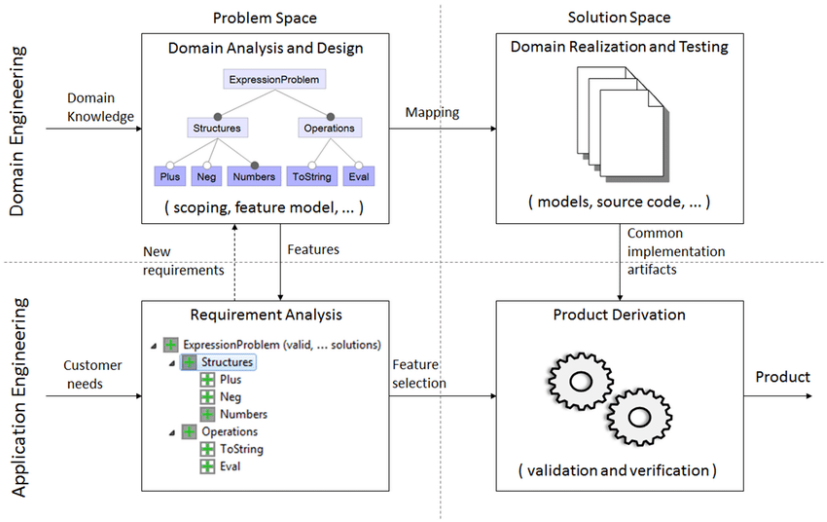


Figure 2.1: Engineering process of software product line [Al-14]

To implement a software product line, there is a need to write the requirements in a modelling language, as discussed previously in 2.1. In our case, a software product line can be represented via feature models[Kan+90] using feature diagrams. This is an easy and a graphical way to represent a large

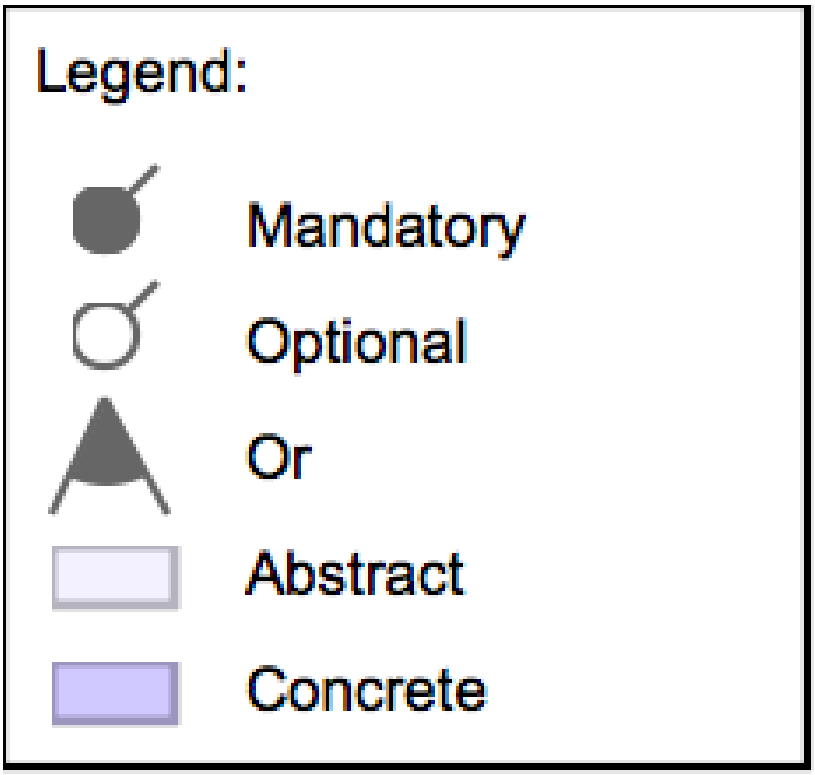


Figure 2.2: Annotations of a Feature Diagram

and complex software product line. Feature diagrams have specific rules and annotations to be used. Those are listed in figure 2.2.

A typical feature diagram of the vending machine would look like figure 2.3.

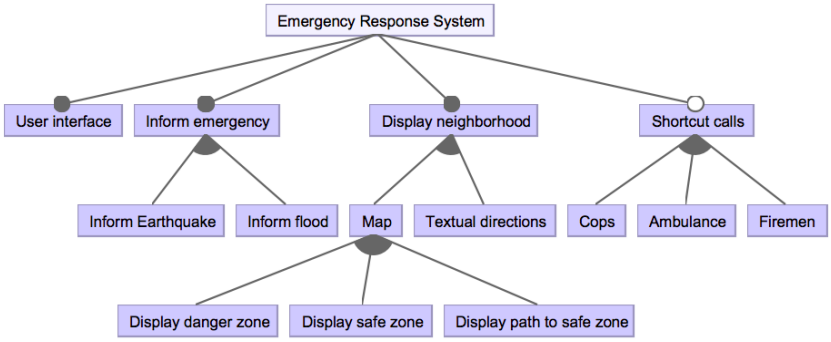


Figure 2.3: Example of a Feature Diagram

Based on the feature diagrams, we see the points of commonality and differences between product configurations. Software product lines can also be represented in propositional logic with formulas such as in figure 2.4.

Feature diagrams are structural: they model the features and their interac-

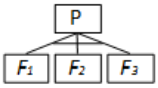
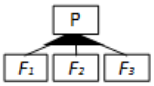
Name	Diagram Notation	Propositional Logic
Mandatory		$P \Leftrightarrow C$
Optional		$C \Rightarrow P$
Exclusive-Or		$(F_1 \Leftrightarrow (\neg F_2 \wedge \neg F_3 \wedge P)) \wedge$ $(F_2 \Leftrightarrow (\neg F_1 \wedge \neg F_3 \wedge P)) \wedge$ $(F_3 \Leftrightarrow (\neg F_1 \wedge \neg F_2 \wedge P))$
Inclusive-Or		$P \Leftrightarrow F_1 \vee F_2 \vee F_3$
Requires	Cross feature arrow	$F_a \Rightarrow F_b$
Excludes	Cross feature arrow	$F_a \Rightarrow \neg F_b \equiv \neg(F_a \wedge F_b)$

Figure 2.4: Example of propositional logic to represent a feature diagram

tions but not their internal behaviours. To model such behaviours, one can use state diagrams.

Regarding the validation part of the life-cycle, researchers had to find a new way to test software product lines without having to cover every possible configuration of the software since this would require too much processing power and might take a very long time to compute, owing to the fact that there are millions of possibilities (see Section2.3).

2.2.1 FeatureIDE

FeatureIDE is a feature-oriented extensible programming tool to eclipse that allows the implementation of software product lines in a structured manner. It has been written in java language and thus supports:

- Feature-oriented programming with AHEAD, FeatureC++ (C++) and FeatureHouse (C),
- Aspect-oriented programming with AspectJ (Java) and FeatureC++ (C++),
- Delta-oriented modeling and programming with DeltaEcore and DeltaJ (Java),
- annotation-based implementation with preprocessor Antenna, C preprocessor CPP by Colligens, and preprocessor Munge.

In the context of this thesis, we have mostly considered using AspectJ for the experimentation instead of the other possibilities due to the reasons below:

- Feature-oriented programming languages had a lack of support for recent java versions. The latest version supported is java 1.4.
- Delta-oriented programming could not be used due to an integration issue with EvoSuite.
- Annotation-based programming languages had a lack of support for the latest java virtual machines. From Antenna's documentation, it seemed like the project had not been continued since 2010.

AspectJ provided the below advantages:

- Supported the latest JDK and hence we could exploit the latest improvements made.
- Could be easily integrated with EvoSuite which is not the case of other approaches.
- Supports all the phases of software product lines development.
- Can easily accommodate changes of software product lines, which happens quite often in this type of structure.

2.3 Testing in Software Product Lines

It is universally known that when implementing software, it is highly recommended to test it in order to ensure the quality of the end product. In the case of software product lines, not only do we have to ensure that the features are written properly but that the different possible combinations (product variations) also integrate correctly. In this section, we will explain the different strategies invented to test software product lines while avoiding the high costs of testing all the possible combinations. There are two strategies to test Software Product Lines:

- Configurations sampling: Configuration sampling approaches samples a representative subset of all the valid configurations of the system and test them individually.
- Variability-aware testing: Variability-aware testing approaches instrument the testing environment to take variability information into account and reduce the test execution effort.

The list below summarises the different testing strategies that exist to test software product lines:

1. Random sampling: This strategy is straightforward. It selects a random subset of the valid configurations.
2. T-wise sampling: T-wise sampling has been derived from Combinatorial Interaction Testing (CIT). The latter relies on the hypothesis that most faults are caused by undesired interactions of a small number of features[KWG04b]. T-wise sampling therefore makes sure that these interactions are covered at least once. The most common t-wise sampling is pairwise. Variations of pairwise are 3-wise, 4-wise and so on.

3. Dissimilarity sampling: This technique approximates t-wise coverage by generating dissimilar configurations (in terms of shared options amongst these configurations). From a set of random configurations of a specified cardinality, a (1+1) evolutionary algorithm evolves this set such that the distances amongst configurations are maximal, by replacing a configuration at each iteration, within a certain amount of time.

4. Incremental sampling: Incremental sampling[Al-+16a] consists of focusing on one configuration and progressively adding new ones that are related to focus on specific parts of the configuration space.

5. One-disabled sampling: The core idea of one-disabled sampling is to extract configurations in which all options are activated but one. The good point is that it makes the algorithm deterministic. However, it implicitly links the bug-finding ability of the algorithm with the solver's internal order.

6. One-enabled sampling: This sampling mirrors one-disabled and consists of enabling each option one at a time. For instance, a configuration where the DisplayNeighbourhood option in figure 2.3 is selected and all the other options are deselected. As for one-disabled, for each selected option, we apply a random selection of the configuration to consider in our evaluation; and the criteria are extended to all-one-enabled, with all the valid configurations for each selected option.

7. Most enabled-disabled sampling: This method only samples two configurations: one where as many options as possible are selected and one where as many options as possible are deselected. If more than one valid configuration is possible for most-enabled (respectively most-disabled) options, we randomly select one most-enabled (respectively most-disabled) configuration. The criteria are extended to all- most-enabled-disabled, with all the valid configurations with most-enabled (respectively most-disabled) options.

There is no sampling criterion dominating others in all cases: it depends on the system and testing goals. However, t-wise coverage demonstrated an excellent bug finding ability while producing small test suites (*e.g.*, [Hal+19]). Therefore, we retain this technique in our work.

2.4 Regression Testing in Software Product Lines

Software product lines typically last for a long time and evolve continuously to address changing requirements. Very often, the end-user's needs change and this has a direct impact on the product. Already derived products often have to be re-derived after such changes to benefit from new and updated features. Programmers thus frequently need to analyze the impact of changes to the software product lines and it's possible configurations to prevent unexpected changes of re-derived products.

Research, so far, has been focusing on either software product lines, testing strategies of software product lines or regression testing[LW89]. While working on this thesis, we did not encounter many research papers that have worked

on the subject of regression testing in software product lines, except for the following ones:

- [RE12]: In this research paper, the author proposes to use a 3d model as an approach to regression testing. The graph has the level(y-axis), version(x-axis) and variant(z-axis) on different axes and this graph can be applied to the 4 approaches proposed in the paper; namely,

    1. Brute Force Strategy – test everything at the domain (commonality) level.
    2. Pure Application Strategy – test everything at application (product) level.
    3. Sample Application Strategy – test a sample at domain level, and the full application testing.
    4. Commonality and Reuse Strategy – test common parts at domain level, and variability at application level.

- [Net+10]: In this research paper, the author has shown a manual way of generating regression test suites in software product lines, integrated in a v-model like approach.

Chapter 3

Software Product Lines Regression Testing

In this chapter, we discuss about the need for regression testing on software product lines and the approach put in place to generate a suite of regression test cases.

3.1 Problem Statement

In this section, we discuss about the issue of regression testing in the context of software product lines. As stated previously, there have been very few research on regression testing in software product lines. We found only 2 studies about it [RE12] [Net+10]. Nevertheless, there is a strong need for exploring this idea. Since, software product lines are bound to evolve over time to accommodate new requirements of the end-user. Every time a change is done on the software product lines, the evolved feature is obviously tested and there is a crucial need to test for any regression that might have been caused due to the change introduced. In the case of a JAVA project, the work of regression testing has been widely documented [Mag+16] [Won+97] [USV14]. However, there are few literature work on regression testing in the context of software product lines.

Our aim is to find a way to generate some regression test cases when the software product lines have evolved in order to detect any faults that might have been caused and propagated in the features.

3.2 Approach

In this section, we address an approach that can be used to generate regression test cases for software product lines that have evolved. We discuss the approach's advantages and drawbacks.

This approach uses a mix of a pairwise algorithm for the generation of configuration samples and some automatic generation of tests in JAVA for features that have changed in the Feature Model. Sampling algorithms have been long used to avoid the explosion of a combinatorial number of product variants. Amongst them, pairwise sampling has been demonstrated [Hal+19] to be the best at providing a reasonable coverage while detecting a maximum number of faults.

Pairwise algorithms can be implemented using the below algorithms:

- ICPL [JHF12],
- Chvatal [Chv79],
- IncLing [Al-+16b],
- Yasa [Kri+20].

In our case, we have used ICPL, which is a fast implementation directly integrated with FeatureIDE.

The approach that is proposed in our research is detailed below.

The steps are:

1. On the core Feature Model,
   - Run a Pairwise Algorithm to derive an initial set of sample product variants.
   - Generate the test cases from the sample product variants.

   Note: For the purpose of this thesis, evolution of feature model has been categorised in 3 main types:
   - Addition of a feature: a new feature is added to the feature model.
   - Deletion of a feature: an existing feature is deleted from the feature model.
   - Modification of a feature: an existing feature of the feature model is modified in its behaviour.

   Any software evolution is a combination of the above 3 categories, our experience was based solely on the each category of evolution. Henceforth, all mentions of evolved feature model in this thesis is referring to an evolution from either of the 3 categories described above.
2. On the evolved Feature Model,
   - Run a Pairwise Algorithm to derive an initial set of sample product variants.
3. Compare the two sample sets, and identify combinations that do not match, which are the ones for which some new unit tests need to be generated. This comparison is illustrated in figure 3.1.
4. Finally, generate the test cases for that subset of sample product variants. This reduces our scope of product variants for which unit test cases need to be generated.

The diagram below summarises the steps:

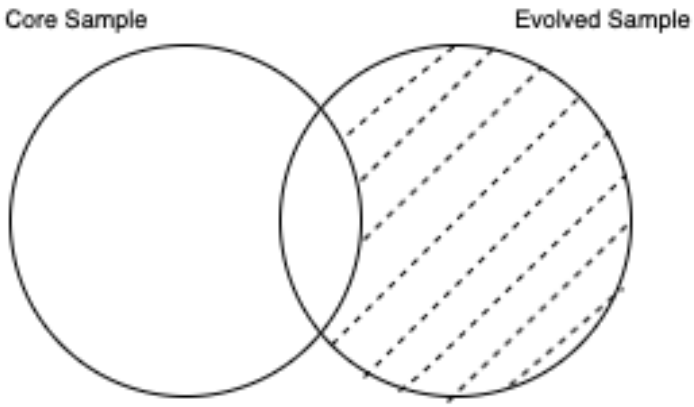


Figure 3.1: Comparison between the core variants and core-evolutions variants. The combinations that do not match are the ones for which some new unit tests need to be generated.

3.2.1 Advantages

The technique consists in a combination of simple steps from pre-defined algorithms. Pairwise sampling is a readily available algorithm that can be applied to any language.

3.2.2 Disadvantages

- Given that the unit tests for the core sample are not available, their production will require processing power and time before even considering the proposed technique. This approach, thus relies heavily on core sample unit tests.
- If this strategy is to be used, the software product line should be in java, since EvoSuite works only with this programming language.
- By definition, the sampling algorithm will not cover everything in the code.

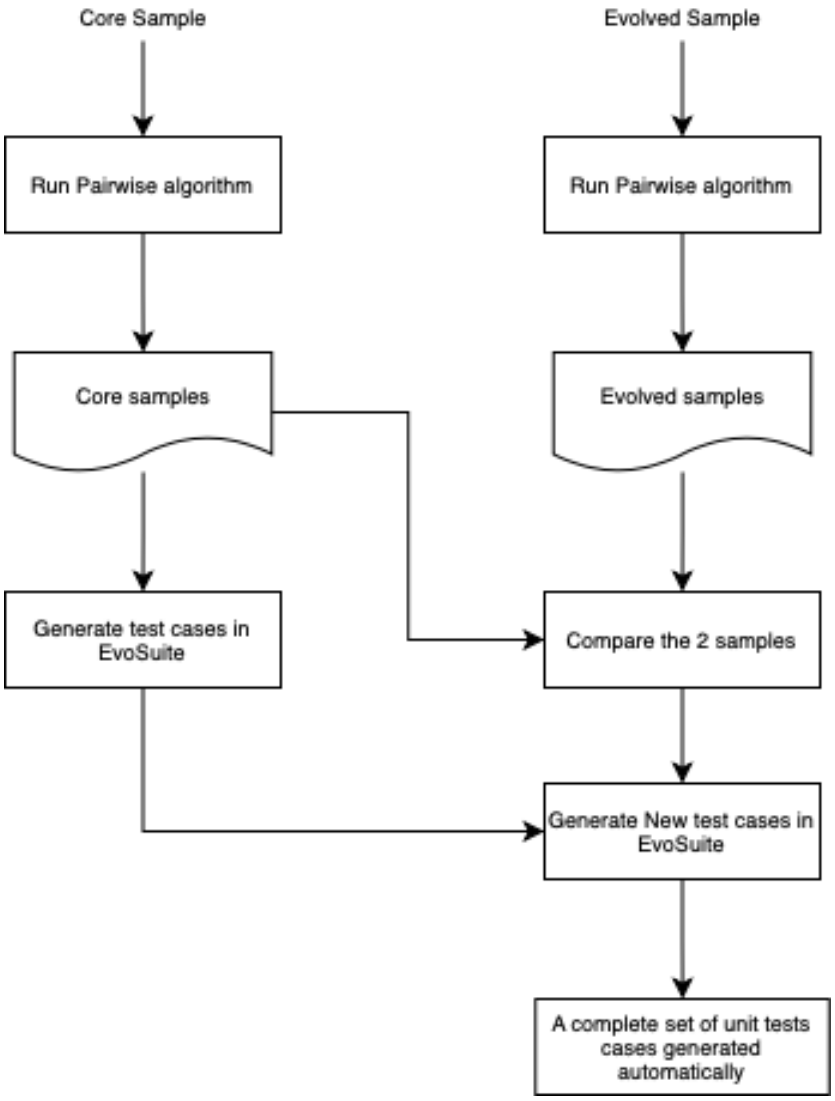


Figure 3.2: Flow chart to summarise the strategy

Chapter 4

Experiment and Discussion

In this chapter, the research questions and the experiment designed to test the automatic generation of regression test cases in the context of software product lines are discussed. We present the results and outline their implications.

4.1 Experimentation

This section explains the research questions and how they were derived. We then describe the experimental setup.

4.1.1 Research questions

Based on the literature review and the research done, two research questions are deducted that this thesis shall cater for.

RQ1: *Can we find a way of automatically generating regression test cases on software product lines?*

Based on our research for the thesis, we have so far, come across two research papers that discuss a way of creating regression test cases in the context of software product lines whereby [RE12] proposes 4 approaches while [Net+10] proposes to have a v-model like approach with manual creation of regression test suite. However, we did not encounter any paper that was able to design an experiment to generate regression test suites in software product lines. Hence, the main contribution of this thesis is to concretely assess the feasibility of generating regression test cases automatically for software product lines. In particular, we consider a core feature model and evolved/combination of evolved feature model according to the strategy presented in Chapter 3.

RQ2: *How efficient is the found method in generating software product line regression test suites?*

A first approach to measure the approach's efficiency is proposed. That is, by manually injecting some faults in the evolved code and assessing if the generated suite is able to uncover them. This will validate the approach and the experiment since discovering bugs in the unchanged part of the code is what a regression test suite is intended for.

4.1.2 Experimentation process

This first experiment's intent is to generate regression test suites on a sample software product lines. For this intent, we chose to use a hand-crafted well known example of the Vending Machine. This approach corresponds to the Sample Application Strategy stated in 2.4. The reason to use this simple example is purely to ensure that the approach works at small scale.

In this experiment, we have used two tools to support our experiment, namely:

- FeatureIDE,
- EvoSuite.

4.1.3 Designing the feature model

As previously described in 2.2.1, FeatureIDE is an extensible plugin on the Eclipse Marketplace that supports the development of software product lines on the Eclipse integrated development environment.

Within the IDE, a FeatureIDE project can be created. From there, the feature model can be designed.

In our case, we have based ourselves, as previously mentioned, on the classic example of the vending machine [Cla+10]. This model is a simple vending machine offering hot and cold drinks and managing payments.

The core feature model designed for the experiment is as shown in figure 4.1.

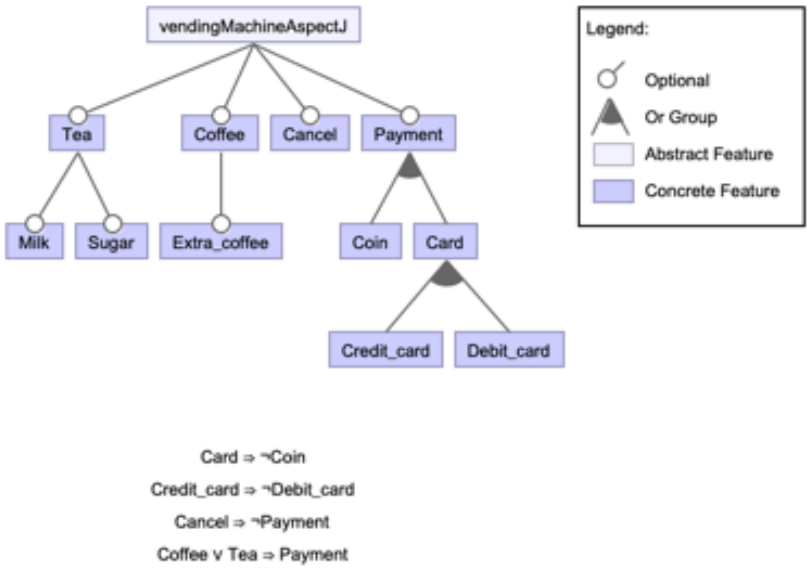


Figure 4.1: Vending machine feature diagram

Based on the feature model in 4.1, the class diagram of the core feature model has been designed as shown in figure 4.2. As one may see, the Vending Machine is the main class used in the experiment, and its features have been implemented

as Aspects [LB02], thanks to the AspectJ integration with FeatureIDE. Aspect-oriented programming is a programming paradigm which focuses on re-usability of cross-cutting concerns. An aspect is an encapsulation of a crosscut in the program (possibly in multiple modules), allowing to alter behaviour at given patterns in the code known as *point-cuts*.

The main method of the class is *action*, which returns a text describing the configuration. A sample returned text might be "The Vending machine is making coffee with sugar and milk for a price of 2 euros". The second most important method is *getTotalPrice* which, based on the accessors *isMilk*, *isSugar* and *isFee*, calculates the total price the customer should pay. The method *unnecessaryComment* returns a joke the vending machine could possibly state to its customer. In our experiment, this last method's purpose is to extend the Vending Machine in order to have a little more features to test later on. As one may see, all the features of the Vending Machine are implemented as aspects, usually *around* the method *action*. Indeed, the Vending Machine's action method will return an empty text and the aspects will concatenate the verb of choosing a feature (e.g. making coffee).

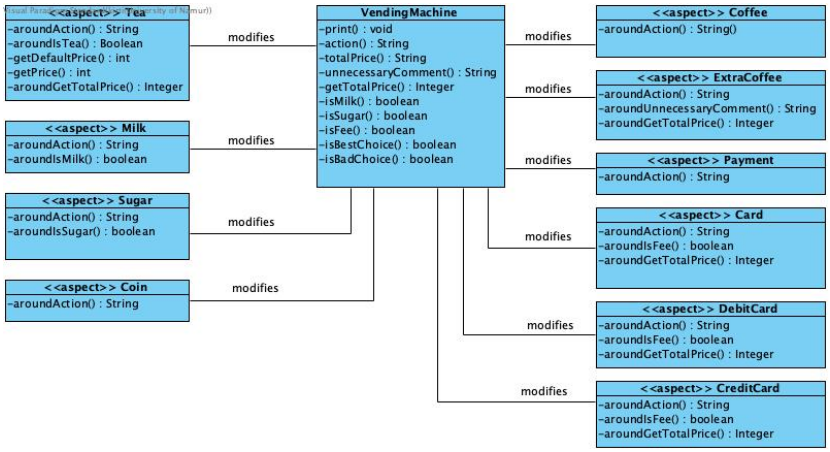


Figure 4.2: Vending machine class diagram

4.1.4 Generating the core test cases

After implementation of the core feature model in AspectJ on FeatureIDE, we derive the sample configurations using the ICPL algorithm. ICPL was chosen since it is one of the main algorithms of generating pairwise samples in [JHF12] for a large-scale system on a similar problem; incremental testing. This algorithm was also used in [Pet+21] for calculating the stability of evolving feature model. Moreover the ICPL algorithm is well integrated in the tool.

To obtain the sample product configurations from the feature model, we used FeatureIDE. After the feature model was designed in the tool as shown in 4.1, we used the *Product Generator* option of FeatureIDE that runs on ICPL algorithm with 2-wise pairings to generate the product configurations as shown in 4.3. We chose to do a 2-wise because it is known to be a quick and efficient

pairing as shown by [KWG04a]. The product configurations are then generated as shown in 4.4.

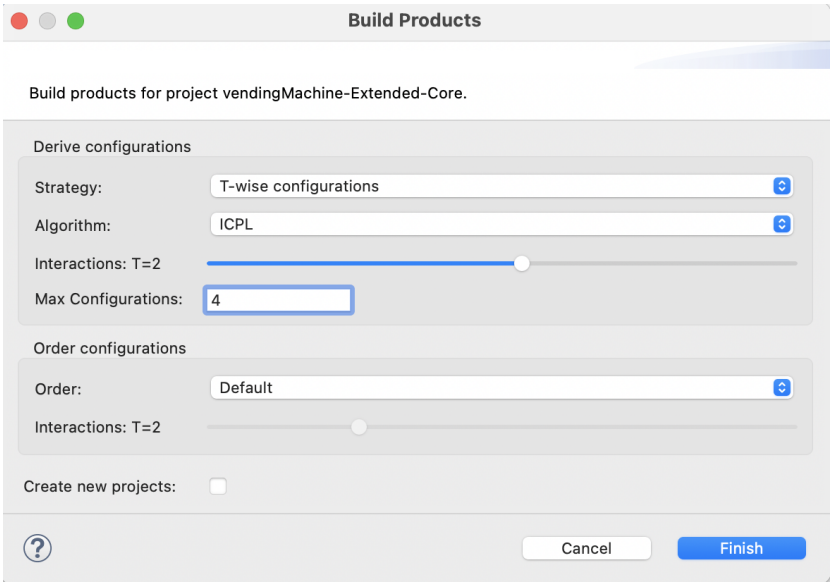


Figure 4.3: Product Generator from FeatureIDE

On each product configuration generated, we used EvoSuite [FA11] to derive the core test cases, which form the basis of our regression test suite. An example of a test suite generated from the product configuration shown in figure 4.5 is given below.

```
import org.junit.Test;
import static org.junit.Assert.*;
import org.evosuite.runtime.EvoRunner;
import org.evosuite.runtime.EvoRunnerParameters;
import org.junit.runner.RunWith;

@RunWith(EvoRunner.class)
@EvoRunnerParameters(mockJVMNonDeterminism = true,
useVFS = true, useVNET = true, resetStaticState = true,
separateClassLoader = true,
useJEE = true)
public class VendingMachine_ESTest extends
VendingMachine_ESTest_scaffolding {

  @Test(timeout = 4000)
  public void test0()  throws Throwable  {
      VendingMachine vendingMachine0 = new VendingMachine();
      vendingMachine0.print();
  }

  @Test(timeout = 4000)
```

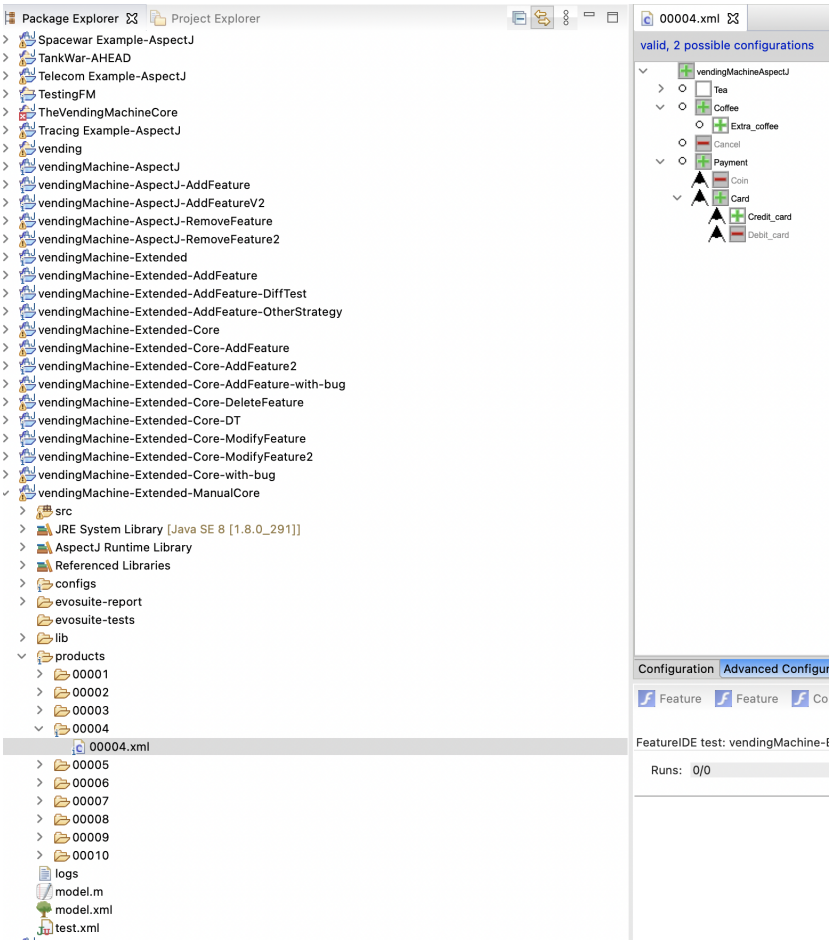


Figure 4.4: Product configurations generated from ICPL

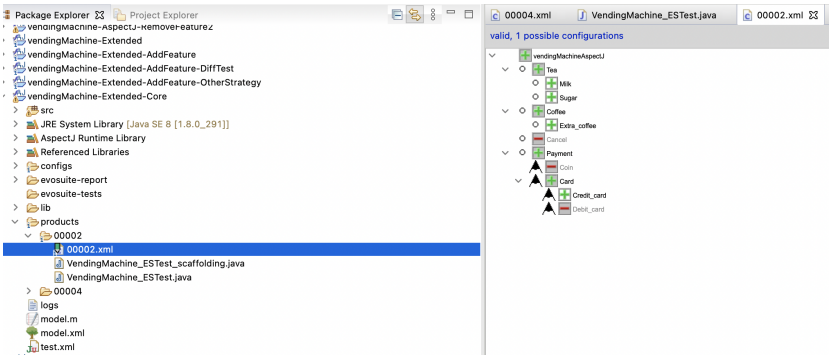


Figure 4.5: Product configuration

```
public void test1()  throws Throwable  {
    VendingMachine vendingMachine0 = new VendingMachine();
    String string0 = vendingMachine0.action();
```

```
        assertEquals("Vending machine is making coffee (and) with
        a card (and) with milk (and) with sugar (and) making tea
        (price : 2) (and) with extra coffee with a credit card
        (adding 1 euro fee) (and) accepting payment", string0);
    }

    @Test(timeout = 4000)
    public void test2()  throws Throwable  {
        VendingMachine vendingMachine0 = new VendingMachine();
        String string0 = vendingMachine0.unnecessaryComment();
        assertEquals("Had a bad night?", string0);
    }

    @Test(timeout = 4000)
    public void test3()  throws Throwable  {
        VendingMachine vendingMachine0 = new VendingMachine();
        Integer integer0 = vendingMachine0.getTotalPrice();
        assertEquals(4, (int)integer0);
    }

    @Test(timeout = 4000)
    public void test4()  throws Throwable  {
        VendingMachine vendingMachine0 = new VendingMachine();
        String string0 = vendingMachine0.totalPrice();
        assertEquals("For a total of 4 euros", string0);
    }

    @Test(timeout = 4000)
    public void test5()  throws Throwable  {
        VendingMachine vendingMachine0 = new VendingMachine();
        boolean boolean0 = vendingMachine0.isMilk();
        assertTrue(boolean0);
    }

    @Test(timeout = 4000)
    public void test6()  throws Throwable  {
        VendingMachine vendingMachine0 = new VendingMachine();
        boolean boolean0 = vendingMachine0.isTea();
        assertTrue(boolean0);
    }

    @Test(timeout = 4000)
    public void test7()  throws Throwable  {
        VendingMachine vendingMachine0 = new VendingMachine();
        boolean boolean0 = vendingMachine0.isSugar();
        assertTrue(boolean0);
    }

    @Test(timeout = 4000)
    public void test8()  throws Throwable  {
```

```
        VendingMachine vendingMachine0 = new VendingMachine();
        boolean boolean0 = vendingMachine0.isFee();
        assertTrue(boolean0);
    }

    @Test(timeout = 4000)
    public void test9()  throws Throwable  {
        String[] stringArray0 = new String[1];
        VendingMachine.main(stringArray0);
        assertEquals(1, stringArray0.length);
    }
}
```

4.1.5 Evolving the feature model

Since our objective is to generate a regression test suite, the next step is to implement the main possible evolution of the core model which we will consider in the regression test suite assessment later.

Those main possible evolution were chosen arbitrarily as:

- The addition of a feature,
- The deletion of a feature,
- The modification of a feature.

In our sample, we chose to implement the addition of a feature *Water*, the deletion of the feature *Coffee* along with *ExtraCoffee*, and the modification of the feature *Tea*, changing the price of the drink. We detail the addition of the feature *Water* below.

Adding a new feature, the evolved feature model is altered and now looks like Figure 4.6.

Again, we generate the sample configurations of the evolved feature model which we may pair with the configurations we generated on the core feature model previously. Indeed, for the addition of a feature, a configuration from the new feature model will be paired to an equivalent configuration in the core feature model without the new feature. For the deletion of a feature, we pair them using the same technique. Finally, for the modification of a feature, we simply pair the exact same configurations. We detail the number of generated configurations from ICPL in table 4.1.

CORE	ADDITION	DELETION	MODIFICATION
10	10	11	10

Table 4.1: Number of configurations generated by ICPL algorithm.

Note: The deletion of a feature generated more configurations since a constraint along with the feature was removed

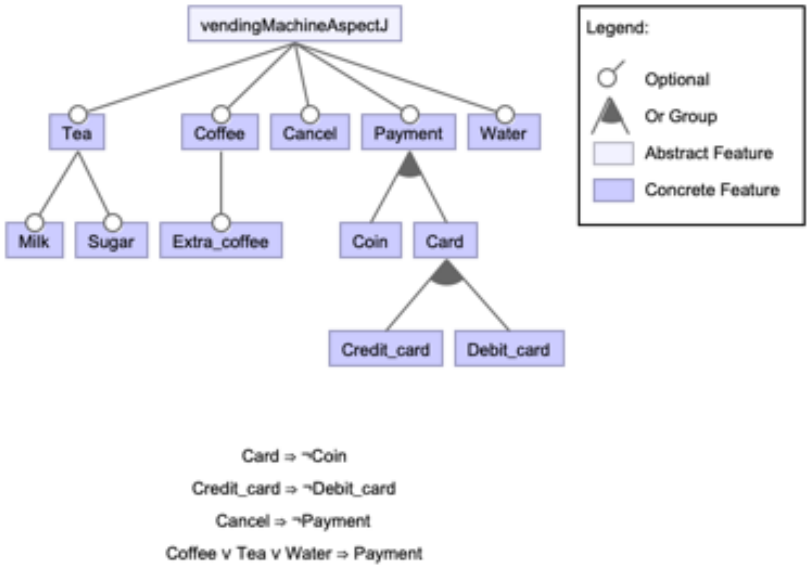


Figure 4.6: Adding the Feature Water - Evolved feature model

4.1.6 Generating the regression test suite

Using the paired configurations discussed in the previous subsection, EvoSuite can be run to automatically generate the regression test suite based on the evolution. That is, using the extension of EvoSuite, commonly known as EvoSuiteR[Sha+13].

An example of the regression test suite generated on a configuration pair in the addition of a feature to the core model is as shown below:

```
import org.junit.Test;
import static org.junit.Assert.*;
import org.evosuite.runtime.EvoRunner;
import org.evosuite.runtime.EvoRunnerParameters;
import org.junit.runner.RunWith;


@RunWith(EvoRunner.class)
@EvoRunnerParameters(mockJVMNonDeterminism = true,
useVFS = true,
useVNET = true,
resetStaticState = true,
separateClassLoader = true,
useJEE = true)
public class VendingMachine_ESTest extends
VendingMachine_ESTest_scaffolding {

  @Test(timeout = 4000)
  public void test0()  throws Throwable  {
```

```
        VendingMachine vendingMachine0 = new VendingMachine();
        Integer integer0 = vendingMachine0.getTotalPrice();

        assertEquals(2, (int)integer0); //
        (Primitive) Original Value: 2 |
        Regression Value: 6
    }
}
```

Such a test was generated by the following command, after selecting the pair configurations as "current configuration" in Feature IDE.

```
$EVOSUITE -class VendingMachine -regressionSuite
-projectCP "<Core model class path>"
-Dregressioncp "<Evolved model class path>"
```

In total, for each configuration pair, EvoSuiteR generated one test suite consisting in one test case, as illustrated previously.

As a summary, the steps taken to design the experiment are:

1. Design the feature model,
2. Implement the feature model,
3. Evolve the core feature model by:
   - Adding a feature,
   - Deleting a feature,
   - Modifying a feature.
4. On each evolution of the feature model against the core feature model run EvoSuiteR to generate the regression test cases.
5. The sum of the regression test cases form the regression test suite.

This last sum may be added to the number of tests EvoSuite generated on the core model as in section 4.1.4. However, those tests have to be manually checked and possibly adapted for the extended feature model in the case of a feature modification and feature deletion.

4.1.7 Assessing the regression test suite

In order to evaluate the efficiency of our regression test suite generation, we make a preliminary experiment by introducing faults ourselves in one of the evolved feature model. That is preliminary following the known approach of systematically introducing faults in the program then assessing how many faults may be discovered using the test suite [MKP18]. That is mutation testing, the mutation score giving the number of faults uncovered divided by the number of faults inserted. In that matter, we introduced logical faults, purposely trying to find leaks in our regression test suite. The number of faults inserted by the type of evolution is given in table 4.2.

Following this approach, we found out that EvoSuiteR did not generate test cases covering the faults in the text returned in the *action* method.

Hence, EvoSuiteR works by

	ADDITION	DELETION	MODIFICATION
Introduced Faults	2	2	2
Uncovered Faults	1	1	1

Table 4.2: Number of faults inserted and uncovered, in each evolved feature model, by its regression test suite.

- Generating a first sample of random inputs,
- Choosing the tests that maximise a fitness function,
- Running those tests against both the initial code (the core feature model in our case) and the evolved code (the evolved feature model) and eliminate the tests that do not produce different outputs,
- Finally, minimising the test suite by removing unnecessary assertions, generated through the traces such as duplicates.

We found out that the minimisation step was probably too restrictive, since it removed assertions that would have found the faults we injected. Indeed, using EvoSuiteR without its minimisation function (-Dminimize=false option), the generated test suite was able to find any of the faults we manually injected. Even though the minimisation function is useful since it helps producing small test suites, easily reusable by a programmer, the function seems to reduce too many assertions. We illustrate this last statement with a sample test case generated by EvoSuiteR without minimisation.

```
import org.junit.Test;
import static org.junit.Assert.*;
import org.evosuite.runtime.EvoRunner;
import org.evosuite.runtime.EvoRunnerParameters;
import org.junit.runner.RunWith;

@RunWith(EvoRunner.class)
@EvoRunnerParameters(mockJVMNonDeterminism = true,
useVFS = true, useVNET = true,
resetStaticState = true,
separateClassLoader = true,
useJEE = true)
public class VendingMachine_ESTest extends
VendingMachine_ESTest_scaffolding {

  @Test(timeout = 4000)
  public void test0()  throws Throwable  {
      VendingMachine vendingMachine0 = new VendingMachine();
      vendingMachine0.isSugar();
      String[] stringArray0 = new String[0];
      VendingMachine.main(stringArray0);
      String string0 = vendingMachine0.action();
      assertEquals("Vending machine is making coffee (and) with a
```

```
card (and) with milk (and) with sugar (and) making tea

(price : 2) (and) with extra coffee with a credit card
(adding 1 euro fee) (and) accepting payment", string0); //
(Primitive) Original Value: Vending machine is making
coffee (and) with a card (and) with milk (and) with sugar
(and) making tea (price : 2) (and) with extra coffee with a
credit card (adding 1 euro fee) (and) accepting payment |
Regression Value: Vending machine is making coffee (and)
with a card (and) with milk (and) with sugar (and) making
tea (price : 2) (and) with extra coffee with a credit card
(adding 1 euro fee) (and) accepting payment (and) pouring
water tea (price : 4)

vendingMachine0.isTea();
String[] stringArray1 = new String[0];
vendingMachine0.isMilk();
vendingMachine0.isMilk();
VendingMachine.main(stringArray1);
String string1 = vendingMachine0.action();

assertEquals("Vending machine is making coffee (and) with a
card (and) with milk (and) with sugar (and) making tea
(price : 2) (and) with extra coffee with a credit card
(adding 1 euro fee) (and) accepting payment", string1); //
(Primitive) Original Value: Vending machine is making
coffee (and) with a card (and) with milk (and) with sugar
(and) making tea (price : 2) (and) with extra coffee with a
credit card (adding 1 euro fee) (and) accepting payment |
Regression Value: Vending machine is making coffee (and)
with a card (and) with milk (and) with sugar (and) making
tea (price : 2) (and) with extra coffee with a credit card
(adding 1 euro fee) (and) accepting payment (and) pouring
water tea (price : 4)

String string2 = vendingMachine0.action();
assertEquals("Vending machine is making coffee (and) with a
card (and) with milk (and) with sugar (and) making tea
(price : 2) (and) with extra coffee with a credit card

(adding 1 euro fee) (and) accepting payment", string2); //
(Primitive) Original Value: Vending machine is making
coffee (and) with a card (and) with milk (and) with sugar
(and) making tea (price : 2) (and) with extra coffee with a
credit card (adding 1 euro fee) (and) accepting payment |
Regression Value: Vending machine is making coffee (and)
with a card (and) with milk (and) with sugar (and) making
tea (price : 2) (and) with extra coffee with a credit card
(adding 1 euro fee) (and) accepting payment (and) pouring
water tea (price : 4)
```

```
Integer integer0 = vendingMachine0.getTotalPrice();
assertEquals(4, (int)integer0); //
(Primitive) Original Value: 4 | Regression Value: 8

vendingMachine0.isFee();
vendingMachine0.isTea();
String string3 = vendingMachine0.totalPrice();

assertEquals("For a total of 4 euros", string3); //
(Primitive) Original Value: For a total of 4 euros |
Regression Value: For a total of 8 euros

Integer integer1 = vendingMachine0.getTotalPrice();

assertEquals(4, (int)integer1); // (Primitive) Original
Value: 4 | Regression Value: 8

Integer integer2 = vendingMachine0.getTotalPrice();

assertEquals(4, (int)integer2); // (Primitive) Original
Value: 4 | Regression Value: 8

vendingMachine0.unnecessaryComment();
vendingMachine0.isMilk();
String string4 = vendingMachine0.action();

assertEquals("Vending machine is making coffee (and) with a
card (and) with milk (and) with sugar (and) making tea
(price : 2) (and) with extra coffee with a credit card
(adding 1 euro fee) (and) accepting payment", string4); //
(Primitive) Original Value: Vending machine is making
coffee (and) with a card (and) with milk (and) with sugar
(and) making tea (price : 2) (and) with extra coffee with a
credit card (adding 1 euro fee) (and) accepting payment |
Regression Value: Vending machine is making coffee (and)
with a card (and) with milk (and) with sugar (and) making
tea (price : 2) (and) with extra coffee with a credit card
(adding 1 euro fee) (and) accepting payment (and) pouring
water tea (price : 4)

vendingMachine0.isSugar();
String string5 = vendingMachine0.totalPrice();

assertEquals("For a total of 4 euros", string5); //
(Primitive) Original Value: For a total of 4 euros |
Regression Value: For a total of 8 euros

vendingMachine0.print();
VendingMachine.main(stringArray0);
```

```
        Integer integer3 = vendingMachine0.getTotalPrice();
        assertEquals(4, (int)integer3);
        // (Primitive) Original Value: 4 | Regression Value: 8

        vendingMachine0.isTea();
        vendingMachine0.isTea();
        vendingMachine0.isSugar();
        Integer integer4 = vendingMachine0.getTotalPrice();
        assertEquals(4, (int)integer4); //
        (Primitive) Original Value: 4 | Regression Value: 8

        vendingMachine0.isTea();
        vendingMachine0.isSugar();
        VendingMachine.main(stringArray0);
        vendingMachine0.unnecessaryComment();
        vendingMachine0.isFee();
    }
}
```

4.2 Discussions

In this section, we present the results obtained from the experiments conducted in 4 by answering the research questions derived in 3.

4.2.1 RQ1: Can we find a way of automatically generating regression test cases on software product lines?

Yes, we may generate regression test cases on software product lines using the EvoSuiteR tool. Hence, the tool generates a first population of random test cases, then runs those on both the core feature model and the evolved feature model test cases. When a different output is detected, then the tool deduces a valid regression test case [Sha+13].

However, the main limitations of this strategy is that in the case of a large program where the classes may contain several thousands of lines of code, which is often the case in the industry, this approach may be costly in computation time. Specially, in the case of software product lines the more complex the model, the more time it will take to run through the whole feature to detect the regression test cases. On our own made software product lines sample, the tool was able to generate the test suite for a given configuration pair in 2 minutes on average. Given that, as seen in table 4.1, the pairwise algorithm generated 10 configurations on average per considered software product lines evolution, a total of 20 minutes is necessary to generate the test suite for one software product lines evolution. Some computation may however be saved by sampling the number of configurations we consider at regression test generation time.

Another point worth mentioning is that the product configuration from the evolved model should be carefully chosen as it might not have selected the evolution brought to the feature model. In the case of a large feature model, this exercise would require some automation. [Pet+21] checked the stability coefficient against evolving feature model but did not consider that the bigger

the change, the less stable the coefficient will be. This part of the reasoning was missing in the research work.

4.2.2 RQ2: How efficient is the found method in generating SPL regression test suites?

In order to validate preliminary the approach and the experiment designed, some faults were manually injected in the evolved code. At first, EvoSuiteR was unable to uncover all the injected faults. However, after removing its minimisation function, the test suites were more efficient, uncovering all the faults we manually injected. Even though we lacked time to elaborate the experiment, this first result is encouraging the usage of EvoSuiteR without its minimisation function and pushes us to improve the minimisation function in a future work.

4.2.3 Threats to validity

In our experiment, a simple software product line was used to derive the test results and it should therefore be validated by several real-life case studies.

Also, the evolution considered were basic ones such as addition, deletion and modification of a feature. In a working software product line, the evolution that the software undergoes maybe a combination of those.

Furthermore, we have used a single software product line in the whole research work. The approach needs to be evaluated on other product lines in different domains, to draw generalisable conclusions.

Chapter 5

Conclusion and Perspectives

In this chapter, we give our conclusion to the work and present future work and improvements.

5.1 Conclusion

The aim of our work is to explore the possibility of regression testing on software product lines. In particular, we use EvoSuiteR and FeatureIDE to model the software product line and generate test cases automatically.

In our work, we propose an approach to derive the regression suite. That is, we use FeatureIDE to model the feature diagram and create the concrete classes for each feature defined. Within this same tool, some basic evolution such as addition of a feature, deletion of a feature and modification of a feature are brought to the feature model. Since most evolution, in real-life, would be a combination of those basic evolution, we consider only the latters in the experimentation and generate the corresponding code. Then, we generate tests using EvoSuiteR which runs test case candidates on the core feature model and compares their output to their run on the evolved feature model, then filters out only the tests that give different output. Finally, we validate our approach on a well-known case model which is the Vending Machine and assess the method by manually injecting faults and challenging the test suites into uncovering those.

5.2 Future work

While working on the research questions, there were several ideas for improvement that could be done to this research topic in order to elaborate the subject.

Since the experiment has been based on a simple yet classic version of the Vending Machine, it can be considered that the same experiment be done on a large-scale real-life software product line. The results of which might be more conclusive and representative of real-life scenarios.

The approach proposed can also be compared to other state-of-the-art strategies or approaches proposed in other papers. This will help to determine the

efficiency and mode of use of our proposed approach.

Furthermore, the experiment designed in this research work could be automated to produce a quantitative amount of results in a shorter time period. This will give more facts and figures onto which we could build a better approach or experiment.

The last point is to evaluate that this method of generating regression test cases is efficient. Hence, to prove that the approach proposed in 3.2 can be deemed valid for a large population of feature model, we should find some metrics that can be used to assess our method efficiency.

A first approach to assess the method's efficiency would be to use GIT mining on actual software product lines projects. Indeed, we could mine the number of bugs that were introduced from one version to another.

Then, we would apply our method on these case studies and check the percentage of bugs EvoSuiteR finds after generating a valid regression test suite.

The advantages of this approach is that it is straightforward and produces a realistic result, that is, based on real world examples. The drawback is that the approach is very time consuming. Also, a possible problem would be assessing the number of actual bugs that are to find between two versions. Another possible problem comes from the nature of our experiment. Hence, a known bug in a given configuration might or might not be a bug in another configuration of the software product lines.

While some preliminary Mutation testing has been done to validate our proposition, this point should be elaborated on a larger scale. The efficiency of our method could be compared with another one by comparing the addition of the Mutation score they obtain for each possible configuration. Indeed, the Mutation score is defined as the number identified faults divided by the number of introduced faults.

Bibliography

[Al-+16a] Mustafa Al-Hajjaji et al. "IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling". In: *SIGPLAN Not.* 52.3 (Oct. 2016), pp. 144–155. ISSN: 0362-1340. DOI: 10.1145/3093335.2993253. URL: https://doi.org/10.1145/3093335.2993253.

[Al-+16b] Mustafa Al-Hajjaji et al. "IncLing: efficient product-line testing using incremental pairwise sampling". In: *ACM SIGPLAN Notices* 52.3 (2016), pp. 144–155.

[Al-14] Ra'Fat Al-Msie'Deen. "Reverse Engineering Feature Models From Software Variants to Build Software Product Lines: REVPLINE Approach". PhD thesis. June 2014.

[Ape+13] Sven Apel et al. "Software product lines". In: *Feature-Oriented Software Product Lines*. Springer, 2013, pp. 3–15.

[Chv79] Vasek Chvatal. "A greedy heuristic for the set-covering problem". In: *Mathematics of operations research* 4.3 (1979), pp. 233–235.

[Cla+10] Andreas Classen et al. "Model Checking ¡u class="uu"¿lots¡/u¿ of Systems: Efficient Verification of Temporal Properties in Software Product Lines". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 335–344. ISBN: 9781605587196. DOI: 10.1145/1806799.1806850. URL: https://doi.org/10.1145/1806799.1806850.

[Con21] Software Product Line Conference. *SPLC Hall of fame*. https://splc.net/fame.html. Accessed: 2021-08-16. 2021.

[FA11] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: Association for Computing Machinery, 2011, pp. 416–419. ISBN: 9781450304436. DOI: 10.1145/2025113.2025179. URL: https://doi.org/10.1145/2025113.2025179.

[FA14] Gordon Fraser and Andrea Arcuri. "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite". In: *ACM Trans. Softw. Eng. Methodol.* 24.2 (Dec. 2014). ISSN: 1049-331X. DOI: 10.1145/2685612. URL: https://doi.org/10.1145/2685612.

[Fra21] Gordon Fraser. *EvoSuite*. https://www.evosuite.org. Accessed: 2021-03-20. 2021.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223. ISBN: 1595930566. DOI: 10.1145/1065010.1065036. URL: https://doi.org/10.1145/1065010.1065036.

[Hal+19] Axel Halin et al. "Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack". In: *Empir. Softw. Eng.* 24.2 (2019), pp. 674–717. DOI: 10.1007/s10664-018-9635-4. URL: https://doi.org/10.1007/s10664-018-9635-4.

[Har+04] M. Harman et al. "Testability transformation". In: *IEEE Transactions on Software Engineering* 30.1 (2004), pp. 3–16. DOI: 10.1109/TSE.2004.1265732.

[HC15] Itti Hooda and Rajender Singh Chhillar. "Software test process, testing types and techniques". In: *International Journal of Computer Applications* 111.13 (2015).

[HM10] Mark Harman and Phil McMinn. "A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search". In: *IEEE Transactions on Software Engineering* 36.2 (2010), pp. 226–247. DOI: 10.1109/TSE.2009.71.

[JHF12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. "An algorithm for generating t-wise covering arrays from large feature models". In: *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 2012, pp. 46–55.

[Kan+90] Kyo C Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[Kri+20] Sebastian Krieter et al. "YASA: yet another sampling algorithm". In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. 2020, pp. 1–10.

[KWG04a] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. "Software fault interactions and implications for software testing". In: *IEEE transactions on software engineering* 30.6 (2004), pp. 418–421.

[KWG04b] D.R. Kuhn, D.R. Wallace, and A.M. Gallo. "Software fault interactions and implications for software testing". In: *IEEE Transactions on Software Engineering* 30.6 (2004), pp. 418–421. DOI: 10.1109/TSE.2004.24.

[LB02] Roberto E Lopez-Herrejon and Don Batory. "Using AspectJ to implement product-lines: A case study". In: *Technical report, University of Texas at Austin* (2002).

[LW89] H.K.N. Leung and L. White. "Insights into regression testing (software testing)". In: *Proceedings. Conference on Software Maintenance - 1989*. 1989, pp. 60–69. DOI: 10.1109/ICSM.1989.65194.

[Mag+16] Cláudio Magalhães et al. "Automatic Selection of Test Cases for Regression Testing". In: *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*. SAST. Maringa, Parana, Brazil: Association for Computing Machinery, 2016. ISBN: 9781450347662. DOI: 10.1145/2993288.2993299. URL: https://doi.org/10.1145/2993288.2993299.

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. "S.P.L.O.T.: Software Product Lines Online Tools". In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: Association for Computing Machinery, 2009, pp. 761–762. ISBN: 9781605587684. DOI: 10.1145/1639950.1640002. URL: https://doi.org/10.1145/1639950.1640002.

[MKP18] Urko Rueda Molina, Fitsum Kifetew, and Annibale Panichella. "Java unit testing tool competition-sixth round". In: *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*. IEEE. 2018, pp. 22–29.

[MP14] Andreas Metzger and Klaus Pohl. "Software product line engineering and variability management: achievements and challenges". In: *Future of Software Engineering Proceedings*. 2014, pp. 70–84.

[Net+10] Paulo Anselmo da Mota Silveira Neto et al. "A Regression Testing Approach for Software Product Lines Architectures". In: *2010 Fourth Brazilian Symposium on Software Components, Architectures and Reuse*. 2010, pp. 41–50. DOI: 10.1109/SBCARS.2010.14.

[Pet+21] Tobias Pett et al. "Stability of Product-Line Samplingin Continuous Integration". In: *15th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS'21. Krems, Austria: Association for Computing Machinery, 2021. ISBN: 9781450388245. DOI: 10.1145/3442391.3442410. URL: https://doi.org/10.1145/3442391.3442410.

[RE12] Per Runeson and Emelie Engstrom. "Software Product Line Testing – A 3D Regression Testing Problem". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, pp. 742–746. DOI: 10.1109/ICST.2012.167.

[Sha+13] Sina Shamshiri et al. "Search-Based Propagation of Regression Faults in Automated Regression Testing". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 2013, pp. 396–399. DOI: 10.1109/ICSTW.2013.51.

[USV14] Sebastian Ulewicz, Daniel Schütz, and Birgit Vogel-Heuser. "Software changes in factory automation: Towards automatic change based regression testing". In: *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*. 2014, pp. 2617–2623. DOI: 10.1109/IECON.2014.7048875.

[Won+97] W.E. Wong et al. “A study of effective regression testing in practice”. In: *Proceedings The Eighth International Symposium on Software Reliability Engineering*. 1997, pp. 264–274. DOI: 10.1109/ISSRE.1997.630875.

Glossary

annotation-based implementation An annotation-based implementation annotates a common code base, such that code that belongs to a certain feature is marked accordingly. During product derivation, all code that belongs to deselected features or invalid feature combinations is removed or ignored to form the final product.. 8

commonality Features that are common across most product configurations. . 7, 11

end-user The stakeholder or client that will benefit from the exploitation of the system built. . 5, 6, 10

EvoSuite An automatic test generation tool that works on java programs. . 4, 5, 9, 15, 18, 20, 24, 25

EvoSuiteR An extension of EvoSuite that covers the part for regression test suite generation. . 24–26, 29–32

feature model A graphical representation of a software product line. . ix, 14, 17–19, 23, 25, 26, 29, 32

Mutation score The mutation score is the percentage of killed mutants divided by the total number of mutants multiplied by 100. . 32

Mutation testing It is a type of testing technique where the programmer injects mutants, also known as faults, in order to see if the test cases can detect them. . 32

regression testing A type of change-related testing to detect whether defects have been introduced or uncovered in unchanged areas of the software. . 13

software development lifecycle A lifecycle covers all the stages of software from its inception with requirements definition through fielding and maintenance.. 3

software product lines Software product lines is the concept of customising standardised software to fit the needs of a client by selecting a set of features from the whole package and adapting it to their needs.. ix, 1–5, 8–11, 13, 17, 18, 29, 31, 32

software testing Software testing is the process of verifying that a software product does what it is supposed to do according to the requirements. The benefits of testing include preventing bugs, reducing maintenance costs and improving performance.. 3

variability The features that change based on the specification of the client. . 5, 9, 11

variant A variant is a valid configuration of the product. . 5, 11